An Aircraft Electric Power Testbed for Validating Automatically Synthesized Reactive Control Protocols

Technical Report

ROBERT ROGERSTEN
KTH Royal Institute of Technology

HUAN XU
California Institute of Technology

NECMIYE OZAY
California Institute of Technology

UFUK TOPCU
University of Pennsylvania

RICHARD M. MURRAY
California Institute of Technology

January, 2013

Abstract

Modern aircraft increasingly rely on electric power for subsystems that have traditionally run on mechanical power. The complexity and safety-criticality of aircraft electric power systems have therefore increased, rendering the design of these systems more challenging. This work is motivated by the potential that correct-by-construction reactive controller synthesis tools may have in increasing the effectiveness of the electric power system design cycle. In particular, we have built an experimental hardware platform that captures some key elements of aircraft electric power systems within a simplified setting. We intend to use this platform for validating the applicability of theoretical advances in correct-by-construction control synthesis and for studying implementation-related challenges. We demonstrate a simple design workflow from formal specifications to auto-generated code that can run on software models and be used in hardware implementation. We show some preliminary results with different control architectures on the developed hardware testbed.

Contents

1 Introduction and Motivation

Aircraft electric power systems have become increasingly important over the years because they support various subsystems and essential services on aircraft. These electrical services and subsystems are commonly referred to as system loads. System loads are of two categories, namely, primary loads (some of these are safety- or mission-critical) and secondary (noncritical) loads. The system needs to ensure that the primary loads are supplied with power at all times; that is, if a fault affects a part of the system that powers a primary load, the system must be able to reconfigure and provide power to the load through another path. In order to reconfigure a system, it is necessary to reroute power, which is accomplished with high power electromagnetic devices called contactors. The contactors are arranged such that they are magnetically held in a preferred state by an applied signal. The state is either open or closed. To reconfigure the contactors to react to faults and modes of operation, the system uses control logic that can sense system conditions and environmental conditions under which the system operates. The electric power system, therefore, includes voltage and current sensors connected to the control logic. In current practice, the control logic is often designed by hand, resulting in lengthy design and verification cycles. As an alternative approach, [1] and [2] explored the application of correct-by-construction reactive controller synthesis techniques.

In this report, we describe our recently developed simulation models and a hardware testbed for validating reactive controllers synthesized using `TuLiP` [1], a temporal logic planning toolbox, in order to investigate the validity of the assumptions made in controller synthesis. `TuLiP` is a collection of Python-based code used for automatic synthesis of correct-by-construction embedded control software. Automatic synthesis of reactive centralized and distributed controllers of aircraft electric power systems is described in detail in [2]. The particular distributed synthesis method adopted in this study is introduced in [3] and [4].

University-scale testbeds for research on correct-by-construction controller synthesis are fairly limited. An advanced diagnostics and prognostics testbed is described in [5]. Some applications of this testbed to the electric power systems of spacecraft and aircraft are detailed in [6]. However, the experiments focused on diagnostic queries of the system, while our work is focused on the implementation of correct-by-construction control protocols for fault-tolerant operations. A robotics testbed implementing correct-by-construction controllers is described in [7].

`TuLiP` can be used to synthesize logic so that the satisfaction of certain safety requirements is guaranteed. The synthesized logic enables the contactors to react to changes in system conditions such as the status of generators and rectifier units. This is commonly referred to as a reactive system. The safety requirements used in our simulation models and hardware testbed

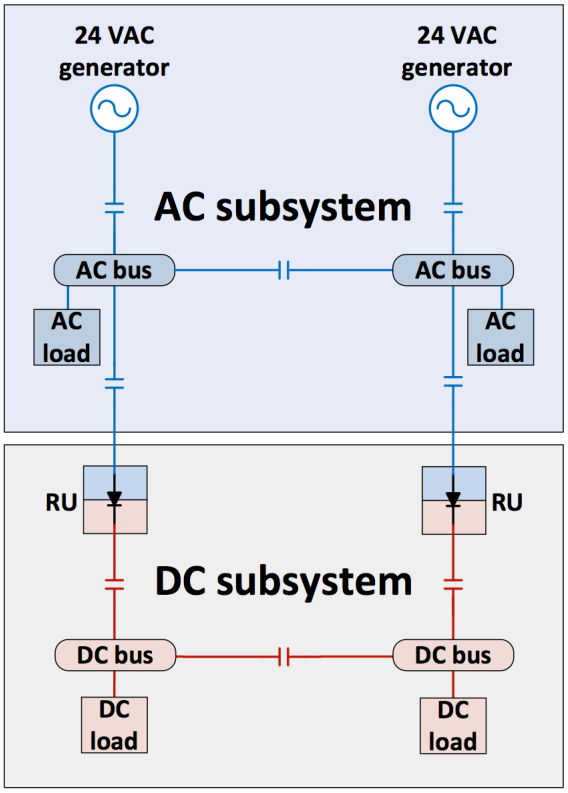


Figure 1: Single-line diagram of the power system testbed. Contactors are represented by double bars. The AC and DC sides of the system are separated by rectifier units (RU).

stipulate that the alternating current generators should never be paralleled and that the duration for which the bus is not powered should never exceed a certain limit. They also include the environment-related assumption that at least a subset of the generators and rectifier units must be working at all times. The simulation models were built with the physical modeling software SimPowerSystems, an extension of Simulink [8]. In order to validate the controller on the experimental hardware platform, we synthesized and tested it using `TuLiP` and SimPowerSystems, respectively. Thereafter, we investigated the validity of the assumptions used for controller synthesis on the experimental hardware platform.

An aircraft electric power system uses different voltage levels, which can broadly be divided into four categories, namely, high-voltage AC, high-voltage DC, low-voltage AC, and low-voltage DC. The topology in Figure 1 is of specific interest because it is representative of some of the key features of aircraft electric power systems in simplified settings. Therefore, the hardware testbed was built based on the above mentioned topology.

2 Theoretical Background

We now discuss the formal specification language utilized for the synthesis of control protocols and how these protocols are implemented in the software models and on the hardware testbed.

2.1 Linear Temporal Logic

In reactive systems, correctness depends, not only on inputs and outputs of a computation, but on execution of the system. Temporal logic is a branch of logic that incorporates temporal aspects to reason about propositions in time. In this report, we consider a version of temporal logic called linear temporal logic (LTL) [9].

LTL includes Boolean connectors like negation ($\neg$), disjunction ($\vee$), conjunction ($\wedge$), material implication ($\rightarrow$), and two basic temporal modalities *next* ($\bigcirc$) and *until* ($\mathscr{U}$). By combining these operators, it is possible to specify a wide range of requirements. Formulas involving other operators can be derived from these basic ones, including *eventually* ($\Diamond$) and *always* ($\Box$).

An *atomic proposition* is a statement on system variables v that has a unique truth value (*True* or *False*) for a given value v. For a set π of atomic propositions, any atomic proposition $p \in \pi$ is an LTL formula. Given a propositional formula describing properties of interest, widely used temporal specifications can be defined in terms of their corresponding LTL formulas as follows. A *safety formula* asserts that a property will remain true throughout the entire execution (i.e., nothing bad will happen). A *response formula* states that at some point in the execution following a state where a property is true, there exists a point where a second property is true. A response formula is used to describe how systems need to react to changes in environment or operating conditions. A response property, for example, can be used to describe how the system should react to a generator failure: if a generator fails, then at some point a corresponding contactor should open [1], [2].

2.2 Reactive Synthesis

A system consists of a set V of variables. The domain of V, denoted by $dom(V)$, is the set of valuations of V. Let E and P be sets of environment and controlled variables, respectively. Let $s = (e,p) \in dom(E) \times dom(P)$ be a state of the system. Consider a LTL specification φ of assume-guarantee form

$$\varphi = \varphi_e \rightarrow \varphi_s,$$

where, roughly speaking, φ_e characterizes the assumptions on the environment and φ_s characterizes the system requirements. LTL formulas are interpreted over infinite sequences of states, where $s_0 s_1 s_2 \ldots$ is an infinite sequence of valuations of environment and controlled variables. The synthesis problem

is then concerned with constructing a strategy, i.e., a partial function $f : (s_0 s_1 \dots s_{t-1}, e_t) \mapsto p_t$, which chooses the move of the controlled variables based on the state sequence so far and the behavior of the environment so that the system satisfies φ_s as long as the environment satisfies φ_e.

For general LTL, the synthesis problem has a doubly exponential complexity [10]. A subset of LTL, namely generalized reactivity (2.2) (GR(1)), can be solved in polynomial time (polynomial in the number of valuations of the variables in E and P) [11]. GR(1) specifications restrict φ_e and φ_s to take the following form, for $\alpha \in \{e,s\}$,

$$\varphi_\alpha := \varphi^\alpha_{\text{init}} \wedge \bigwedge_{i \in I^\alpha_1} \Box \varphi^\alpha_{1,i} \wedge \bigwedge_{i \in I^\alpha_2} \Box \Diamond \varphi^\alpha_{2,i},$$

where $\varphi^\alpha_{\text{init}}$ is a propositional formula characterizing the initial conditions; $\varphi^\alpha_{1,i}$ are transition relations characterizing safe, allowable moves and propositional formulas characterizing invariants; and $\varphi^\alpha_{2,i}$ are propositional formulas characterizing states that should be attained infinitely often. For the specifications considered in this report, the safety fragment of GR(1) suffices.

Given a GR(1) specification, the digital design synthesis tool implemented in JTLV (a framework for developing temporal verification algorithm) [12] generates a finite-state automaton that represents a switching strategy for the system. `TuLiP` provides an interface to JTLV.

2.3 Testbed Specifications

Consider the single-line diagram in Figure 1 in which environment variables are health statuses of generators and rectifier units, and controlled variables are the state of contactors. Consider also two different controller implementations: a *centralized logic* that runs the system with a single automaton and a *distributed logic* that has two different automata, one for the AC subsystem and one for the DC subsystem, running sequentially.

For the centralized logic, the environment assumptions are: (i) at least one generator must always be healthy, and (ii) at least one rectifier unit must always be healthy. In LTL, this can be written as

$$\begin{aligned} \Box(((gen_1 = healthy) \vee (gen_2 = healthy)) \wedge \\ ((ru_1 = healthy) \vee (ru_2 = healthy))), \end{aligned} \tag{1}$$

where gen_1, gen_2, ru_1, and ru_2 are health statuses of the two generators and the two rectifier units, respectively. To ensure non-paralleling of AC sources, we disallow any configuration of contactors in which a path may be created between the two generators. The contactors c_1 and c_2 are below the generators in Figure 1, and c_3 is between the AC buses. Therefore, contactors c_1, c_2, and c_3 can never be closed at the same time. This is written as

$$\Box\neg((c_1 = closed) \wedge (c_2 = closed) \wedge (c_3 = closed)).$$

The last specification ensures that all buses can be unpowered for no more than a time T. The limit that unpowered time can be set to depends on timing characteristics of the testbed, witch is explained in Section 4.1. To synthesize centralized logic, we used the assumption that this time is zero; thus, the specifications that all buses b_i fulfill $\square(b_i = \mathit{powered})$, for $i \in \{1,2,3,4\}$ can be set.

To synthesize distributed logic, we separate the system into two subsystems, seen in Figure 1. The AC subsystem contains all AC components (generators, AC contactors, AC buses, and loads). The DC subsystem contains all rectifier units, DC contactors, buses, and loads. All specifications from the centralized case decompose and carry over to the distributed case. However, in order to ensure that the overall specification is realizable, we impose additional restrictions on the components located at the interface between subsystems.

The rectifier units contain capacitors that can be chosen so that they create a delay T_{RU}, in which the DC buses stays powered even after that an AC bus gets unpowered.

If $T_{RU} > T$ the additional interface refinement comes in the form of a guarantee specification that all DC buses b_i, for $i \in \{1,2\}$ will always be powered $\square(b_i = \mathit{powered})$, provided that both rectifier units stay healthy, i.e.,

$$\square((ru_1 = \mathit{healthy}) \wedge (ru_2 = \mathit{healthy})).$$

This guarantee is written as an environment for the DC subsystem. With this refinement, both subsystems can be synthesized independently, and the overall system specifications are satisfied when they are implemented together. We assume that the time a generator remains healthy is not arbitrarily short so that the AC bus powered time (i.e., the time between two intervals when AC bus is unpowered) is large enough to keep the capacitors on rectifier units charged.

2.4 Implementing Formal Specifications

`TuLiP` generates finite-state automata in the form of a text file that enumerates the possible states of the system and how the transitions could be carried out according to the current state. It also generates a text file that specifies environment variables (e.g., generators and rectifier units) and system variables (e.g., contactors). In order to implement the control logic in SimPowerSystems, we automatically translate these files into a Matlab-compatible script. A preliminary solution uses a Python script for this translation. A Python script generating the Matlab code is released with `TuLiP` version 0.3c under the tools directory[1].

[1] `http://tulip-control.sf.net`

```
State 0 <gen1:1, gen2:1, c1:1, c2:1, c3:0>
With successors: 1, 2, 3, 0
State 1 <gen1:0, gen2:0, c1:0, c2:0, c3:0>
With no successors
State 2 <gen1:0, gen2:1, c1:1, c2:0, c3:1>
With successors: 1, 2, 3, 0
State 3 <gen1:1, gen2:0, c1:0, c2:1, c3:1>
With successors: 1, 2, 3, 0
```

Figure 2: Sample of a `TuLiP` output in two-generator and three-contactor case. The generator status variables are gen1 and gen2, and the contactor status variables are c1, c2, and c3. Each state has successors, which define where the controller can transit depending on current state. In addition, no-successor states exist.

```
function [c1, c2, c3] = mscript(gen1, gen2)
global state;
switch (state)
case 0:
  if gen1 == 1 and gen2 == 1 then
    state = 0; c1 = 1; c2 = 1; c3 = 0;
  else if gen1 == 0 and gen2 == 0 then
    state = 1; c1 = 0; c2 = 0; c3 = 0;
    ...
  end if
case 1:
  ...
end switch
```

Figure 3: Sample code generated using `TuLiP` controller shown in Figure 2.

Figure 2 shows an example four-state `TuLiP` generated controller for the two-generator and three-contactor case. A few lines of the auto-generated code that corresponds to this controller is shown in Figure 3. The auto-generated code can be inserted in SimPowerSystems as a Matlab function block. It can also be connected to the board with the code shown in Figure 4. The complete version of the code in Figure 4 is given in Appendix A. An example of how a SimPowerSystems model can look like is shown in Figure 5.

```
global state;
while 1 do
  gen1 = readgen1();
  gen2 = readgen2();
  [c1, c2, c3] = mscript(gen1, gen2);
  writeboard(c1, c2, c3);
end while
```

Figure 4: Code that implements the control software running on hardware model.

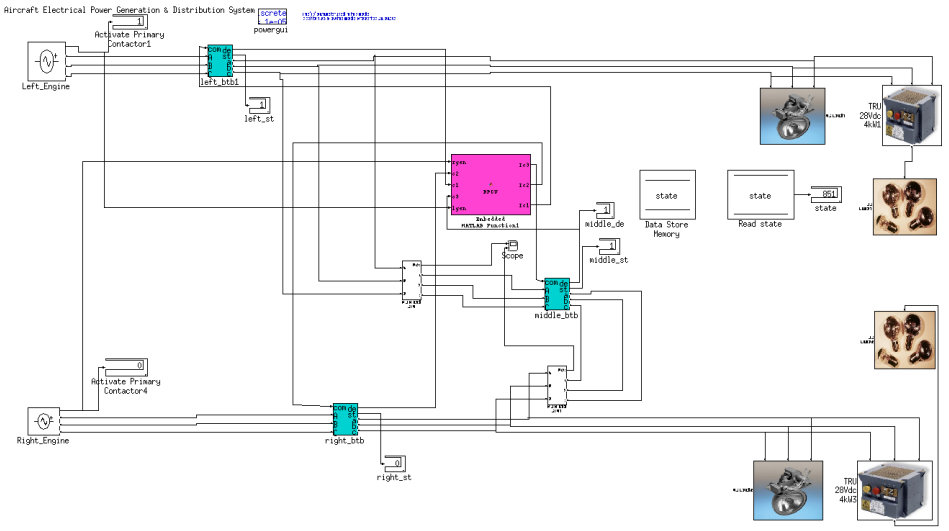


Figure 5: An example of how a SimPowerSystem model can look like. This model corresponds to the AC subsystem shown in Figure 1.

3 Design and Implementation

The single-line diagram in Figure 1 is a simplified notation for representing a three-phase power system. However, as described in Section 3.1, power supply to the hardware testbed is not three-phase. In order to represent the installations of the sensors, circuit protection devices, and fault injection switches, we present a detailed schematic of the testbed in Figure 6. Descriptions of the components shown in Figure 6 are given in Figure 7.

The hardware testbed has two different voltage levels: 24 VAC and 2.5 VDC. The DC section is connected to the AC section by rectifier units. Aircraft contactors are designed to switch three-phase electric power with relatively high currents. Relays are generally used for switching lower currents. These operate in a similar fashion to contactors but are lighter, simpler, and less expensive. Therefore, it was more convenient to handle the switching in the hardware model with relays. It was possible to connect the control logic to the relays with the use of a relay board[2], which is a set of computer-controlled relays that can communicate with programming languages supporting serial communications, e.g., MATLAB. Analog-to-digital (A/D) connections on the relay board are used to monitor the system conditions. A photo of the setup[3] is shown in Figure 8. The transformers in Figure 8 are connected to power cords; these can be unplugged to simulate a generator failure. The rectifier units are connected to a switch, which can be used to generate a fault on the DC subsystem. An explicit component list for the hardware testbed is listed in Appendix B. Next, we describe how we monitor and sense the status of generators and rectifier units.

3.1 Generation and Circuit Protection

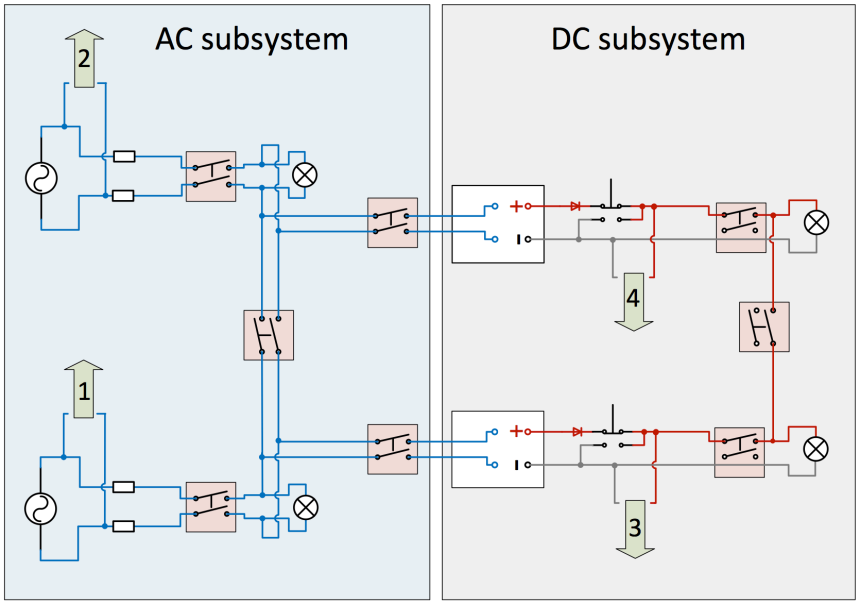
Each generation unit consists of a 12 V battery connected to an inverter that generates 120 VAC; that is then transformed down to 24 VAC to ensure safety. If the controller violates one of the safety requirements and connects these two sources in parallel, it would result in a short-circuit and cause the fuses installed next to the generators, shown in Figure 6(a), to blow. This observation makes it possible to monitor the correctness of the controllers at run time.

3.2 Sensing

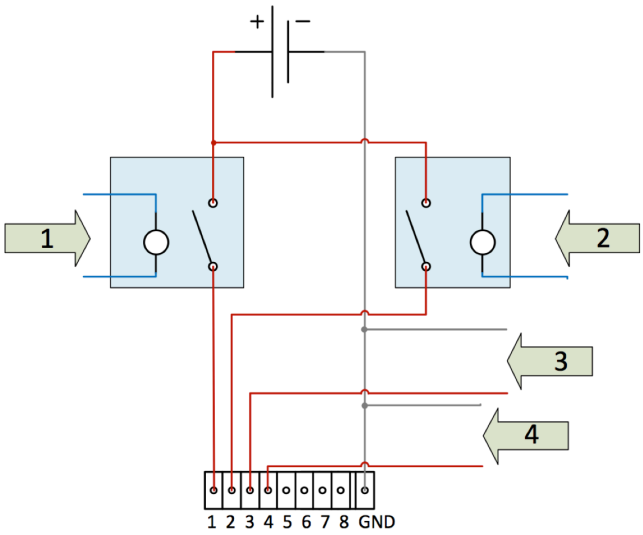
The relay board needs to react consistently to faults injected into the system; this requirement implies that sensor placement, functionality, accuracy, and

[2]A company called RelayPROS sells such relay boards. For more information, visit `www.relaypros.com`.

[3]A photo of the relay board can be found online at `assets.controlanything.com/photos/usb_relay/ZADSR165DPDTPROXR_USB-900.jpg`



(a) Circuit schematic



(b) Sensing configuration

Figure 6: Circuit schematic of the hardware testbed, which corresponds to the single-line diagram shown in Figure 1. The numbered arrows in (**a**) denote voltage sensing connections to the corresponding numbered arrows in (**b**).

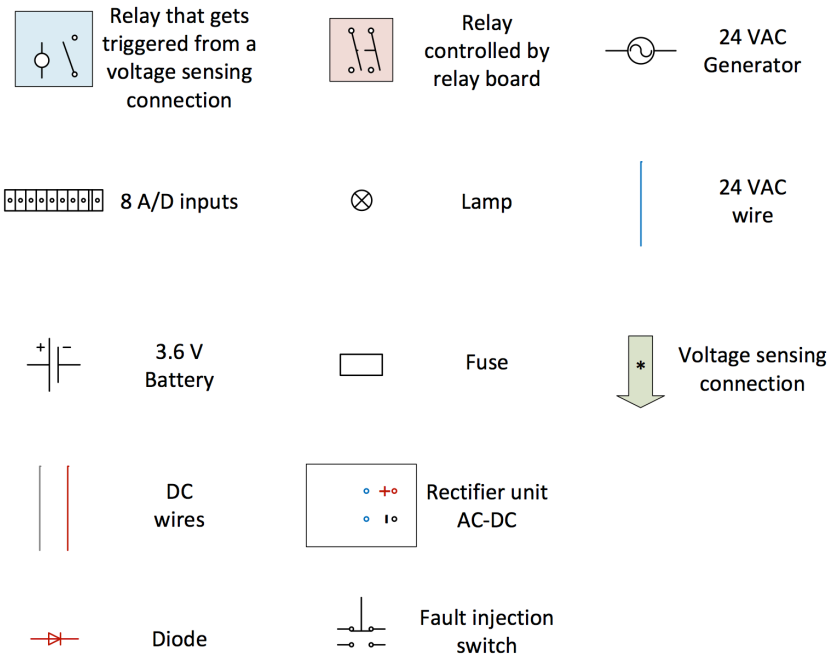


Figure 7: Description of the components used in Figure 6.

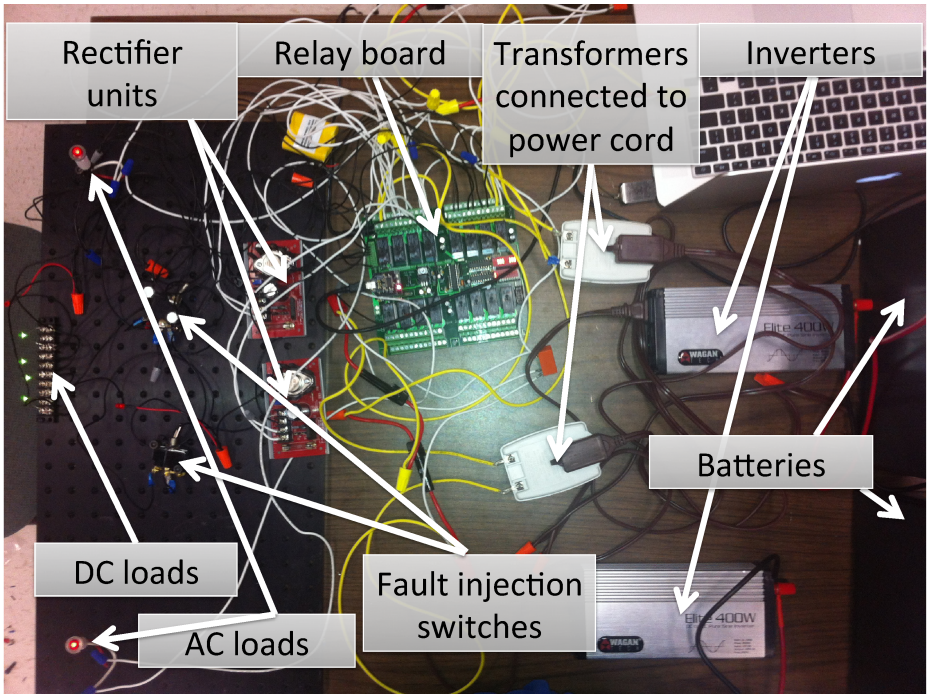


Figure 8: Hardware setup corresponding to the single-line diagram shown in Figure 1.

time delay play crucial roles in design. Two types of faults can be injected in the system, namely, rectifier unit failures and generator failures. Voltage sensing for generator failures is handled using additional relays. These relays close a 3.6 V circuit to a battery when triggered by the voltage from the transformers. If a fault occurs and a generator does not work properly, the 3.6 V circuit opens and the system reacts accordingly. The voltage sensors of the rectifier units are directly connected to the A/D ports of the relay board because the voltage can be tuned to the appropriate value using an adjustable output on the rectifier units. Figure 6(b) illustrates the sensing configuration.

4 Experiments

We next describe the characteristics of the hardware testbed and show some preliminary test runs with different control architectures.

4.1 Testbed Characteristics

The first step before the implementation and testing of different controllers is characterizing the timing properties of the hardware testbed. Every relay has a time delay between the time a command is sent by the computer and the time an action (i.e., relay opening or closing) is taken, this is referred to as the *relay delay time*, T_d. Furthermore, the system has delays resulting from *control cycle times*, T_c and T_c', defined as

$$\begin{aligned} T_c &= T_r + T_I + T_w \\ T_c' &= T_r + T_I, \end{aligned} \tag{2}$$

where T_r is the time it takes to read the health statuses from all of the four environment variables, T_I is the time it takes to run the logic (the time can be interpreted as the time taken to run the code shown in Figure 3), and T_w is the time it takes to write information to the board (see Figure 4). Writing information to the board is not needed in every iteration (for instance, if the system state remains the same), therefore the control cycle time also include T_c'.

The control cycle times T_c and T_c' are listed in Table 1. The relay delay time can be found from the board specifications and shall be less than 20ms.

An important safety requirement in an aircraft is that a bus should never lose power for more than a certain duration, e.g., typically 50ms. In the hardware testbed, the time for which the bus is unpowered depends on the control cycle times and the relay delay time, and because the control cycle times exceed 50ms, we cannot use the typically specified time for which an aircraft can be unpowered. Therefore, it was necessary to adopt a suitable limit. As illustrated with two environment variables in Figure 4 the relay board read the health status from each environment variable in a specified order. It is therefore necessary to include a part of T_c' from the previous control cycle in this limit. The time T_I in Equation (2) is negligible compared

	T_c [ms]	T_c' [ms]
Mean	303.7	187.5
Max	333.3	234.1
Min	282.5	166.6

Table 1: Control cycle time, both when relay configuration changes, i.e., T_c and without any change, i.e., T_c'. The values with and without change were calculated from 20 and 250 measurements, respectively.

to T_r and T_w, the time taken to read the health status from one environment variable can therefore be approximated as $T_c'/4$. A reasonable value of an acceptable unpowered time for the hardware testbed can be

$$T \approx \max(T_d) + \max(T_c) + \frac{4-n}{4}\max(T_c'), \tag{3}$$

where $n \in \{1,2,3,4\}$ is the number which denotes the order of when the environment variable that is faulty is read in the code.

4.2 Controller Tests

Two controllers were tested, one with distributed logic and one with centralized logic. The controller with centralized logic had a 16-state automaton synthesized as explained in Section 2.3. The controller with distributed logic had two four-state automata that run on each subsystem. Both of these automata were synthesized in a similar fashion to the 16-state controller.

If the environment-related assumption is violated, the controller may end up in a state with no outgoing transitions, referred to as the *no-successor* state. The environment-related assumptions for the testbed are expressed in Equation (1) of Section 2.3. A violation of Equation (1) results in the controller entering a no-successor state, which happens when both generators or both rectifier units are faulty. If a centralized controller senses that both rectifier units are faulty, the whole system stops working because a no-successor state has been reached. This is not the case when distributed logic is used, because the AC system continues working even if the DC environment assumption is violated and the DC part reaches a no-successor state. The distributed logic implementation has two different automata that represent the logic, one for each subsystem, with coupling between them. However, the distributed logic is centralized in that it consists of single control software running on a single computer and communicating with the hardware through a single channel.

Figure 9 shows the voltage measurement for the centralized 16-state controller. The measurement was taken on the AC bus when the generator, which health status is read at second place ($n = 2$ in Equation (3)) of the four environment variables in the code, was switched off and then on again. The generator was switched off at $t = 2.83$s, at which point the bus becomes unpowered. The second vertical line from the left indicates when the controller reacts and power up the bus using the other generator, which happens at $t = 3.1$s. The generator was switched on again at $t = 3.73$s; this was accompanied by a discernible change in the sine curve. Once a generator is switched on again after a fault, the time for which the bus is without power is not noticeable because the controller sends simultaneous commands to two relays.

The measured bus-unpowered times are listed in Table 2, which show

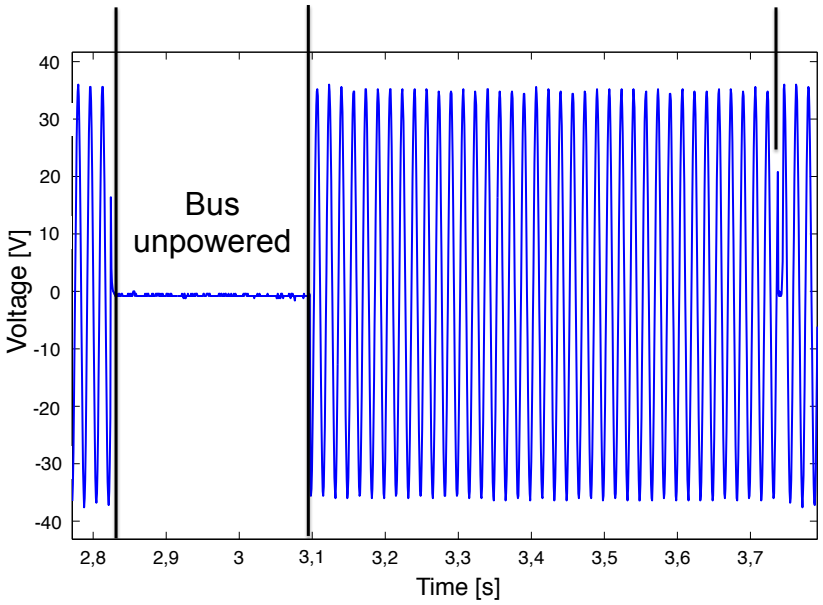


Figure 9: Bus voltage measurement when a generator is switched off and then turned back on. The first vertical line indicates the fault, the second vertical line is when the controller reacts, and the third line is when the generator is turned back on.

	Bus-unpowered time [ms]
Mean	333.9
Max	414.9
Min	232.7

Table 2: Time for which bus is unpowered after a fault is injected. These values are calculated using measurements from 10 fault injections.

a maximum value of $T_{max} = 414.9\text{ms}$. An acceptable unpowered time when $n = 2$ and $\max(T_d) = 20\text{ms}$ can be calculated with Equation (3). It follows that $T \approx \max(T_d) + \max(T_c) + \frac{1}{2}\max(T_c') = 470.35\text{ms}$ and hence, $T_{max} < T$. We used a digital storage oscilloscope (Rigol DS1052E 50MHz) for measurements. The measurement data are imported into MATLAB to plot sinusoidal curves (e.g., Figure 9) and to analyze the signal to estimate the unpowered times.

5 Limitations and Extensions

As discussed earlier, when we implemented distributed logic with the hardware model, it was still centralized in that only one relay board was connected to one computer. However, it is possible to use two relay boards connected to two different computers, with each of them controlled by different automata. The part that contributes the most to the control cycle times (T_c and T_c'), is the time it takes to read data from the board (T_r); if the controller is operated with two relay boards, T_r would be split in half, which would cause T_c and T_c' to decrease significantly. The distributed control architecture would also be more like that of an aircraft.

We injected faults in the hardware testbed by unplugging the power cords and changing the switches; however, a more accurate approach to generate faults would be the use of an additional relay board. Using an additional fault injection board, we can systematically study synchronous, correlated, and cascaded failures and their influence on controller performance; with the current method of fault injection, it could be difficult to switch off a generator and a rectifier unit within the same control cycle.

On an aircraft, the controller is an embedded system designated for a specific task. To increase its reliability and performance, the hardware model could be adapted to run the relay boards through microcontrollers. Embedded code for these microcontrollers can be readily generated using MATLAB.

At last, we want to emphasize the fact that it is entirely possible to synthesize the controller with another synthesis tool and test it on the testbed. It was convenient as an initial demonstration to choose LTL, reactive synthesis, and `TuLiP` because they have been applied to electric power systems in the past [2].

6 Acknowledgments

The authors acknowledge the funding from MuSyC, the Boeing Corporation, and AFOSR (award # FA9550-12-1-0302), and thank Rich Poisson from Hamilton-Sundstrand for helpful discussions about the development of the hardware testbed.

7 References

[1] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. Murray, "`TuLiP`: a software toolbox for receding horizon temporal logic planning," in *International Conference on Hybrid Systems: Computation and Control*, 2011.

[2] H. Xu, U. Topcu, and R. Murray, "A case study on reactive protocols for aircraft electric power distribution," in *IEEE Conference on Decision and Control*, 2012.

[3] N. Ozay, U. Topcu, and R. M. Murray, "Distributed power allocation for vehicle management systems," in *IEEE Conference on Decision and Control*, 2011.

[4] N. Ozay, U. Topcu, T. Wongpiromsarn, and R. M. Murray, "Distributed synthesis of control protocols for smart camera networks," in *ACM/IEEE International Conference on Cyber-Physical Systems*, (Chicago, IL), 2011.

[5] S. Poll, A. Patterson-hine, J. Camisa, D. Garcia, D. Hall, C. Lee, O. J. Mengshoel, D. Nishikawa, J. Ossenfort, A. Sweet, S. Yentus, I. Roychoudhury, M. Daigle, G. Biswas, and X. Koutsoukos, "Advanced diagnostics and prognostics testbed," in *International Workshop on Principles of Diagnosis*, pp. 178–185, 2007.

[6] O. J. Mengshoel, A. Darwiche, K. Cascio, M. Chavira, S. Poll, and S. Uckun, "Diagnosing faults in electrical power systems of spacecraft and aircraft," in *Innovative Applications of Artificial Intelligence Conference*, (Chicago, IL), pp. 1699–1705, 2008.

[7] M. Lahijanian, M. Kloetzer, S. Itani, C. Belta, and S. B. Andersson, "Automatic deployment of autonomous cars in a robotic urban-like environment," in *IEEE International Conference on Robotics and Automation*, (Kobe, Japan), pp. 2055–2060, 2009.

[8] Simulink, *version 7.7 (R2011a)*. Natick, Massachusetts: The MathWorks Inc., 2011.

[9] C. Baier and J. Katoen, *Principles of Model Checking*. MIT press, 1999.

[10] A. Pnueli and R. Rosner, "Distributed reactive systems are hard to synthesize," in *IEEE Symposium on Foundations of Computer Science*, 1990.

[11] N. Piterman, A. Pneuli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *Verification, Model Checking and Abstract Interpretation*, vol. 3855, 2006.

[12] A. Pnueli, Y. Sa'ar, and L. Zuck, "*JTLV* a framework for developing verification algorithms," in *International Conference on Computer Aided Verification*, 2010.

A Code to Implement Control Software

The code below shows how the control software can be implemented on the hardware testbed when the board from RelayPROS is used. The numbers on the relays in Figure 10 are the same as they have to be connected when controlling the board through MATLAB with the code below. The numbers range from 0 to 7, i.e., eight relays in total for each relay bank. There are two relay banks on the relay board, i.e., 16 relays in total. The models built in this report only use Relay Bank 1.

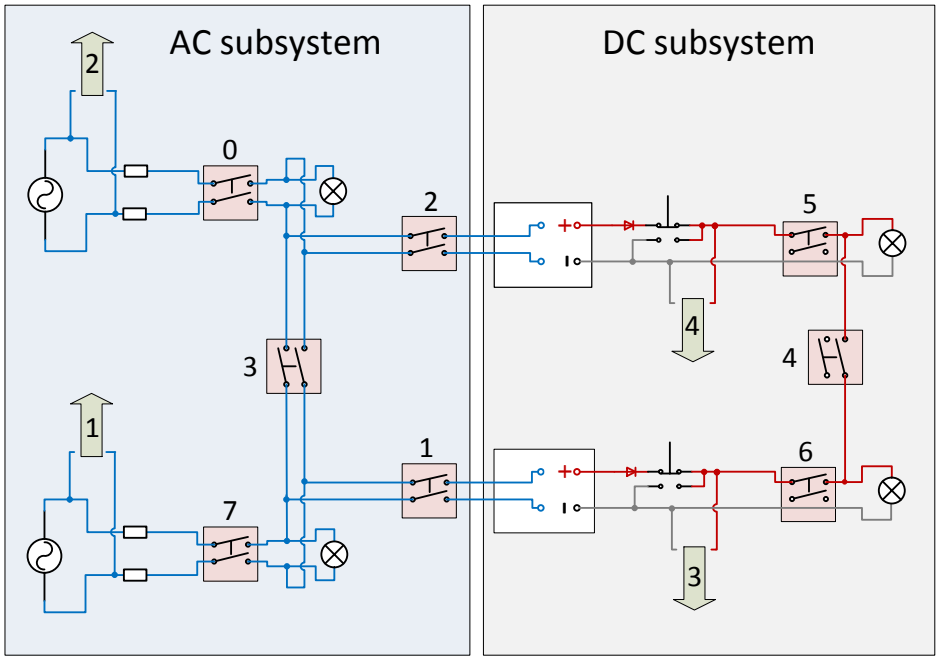


Figure 10: Circuit schematic of the hardware testbed, which corresponds to the single-line diagram shown in Figure 1.

```
clc
clear
global state;
global stateDC;
state = 0;
stateDC = 0;
%choose the correct COM port for the computer in use.
s=serial('COM3');
set(s,'BaudRate',115200)
fopen(s)
%Set initial status
fwrite(s,254)
fwrite(s,8)
fread(s,1);
fwrite(s,254)
fwrite(s,15)
fread(s,1);
fwrite(s,254)
fwrite(s,3)
fread(s,1);
fwrite(s,254)
fwrite(s,9)
fread(s,1);
fwrite(s,254)
fwrite(s,10)
fread(s,1);
fwrite(s,254)
fwrite(s,4)
fread(s,1);
fwrite(s,254)
fwrite(s,14)
fread(s,1);
fwrite(s,254)
fwrite(s,13)
fread(s,1);
%read relay status
fwrite(s,254)
fwrite(s,16)
c2a = fread(s,1);
fwrite(s,254)
fwrite(s,23)
c1a = fread(s,1);
fwrite(s,254)
fwrite(s,19)
c3a = fread(s,1);
fwrite(s,254)
fwrite(s,17)
c4a = fread(s,1);
fwrite(s,254)
fwrite(s,18)
c5a = fread(s,1);
fwrite(s,254)
fwrite(s,20)
c6a = fread(s,1);
disp('[c1,c2,c3,c4,c5,c6] =')
disp([c1a,c2a,c3a,c4a,c5a,c6a])
statecount=0;
count=1;
pause(1)
disp('start')
disp('----------------------------------------')
disp('system runs normal')
disp(state)
%If distributed control logic is used
%uncomment the following row
%disp(stateDC)
while count<100
    tic
    fwrite(s,254);
    fwrite(s,150);
    leftg = fread(s,1);
    fwrite(s,254);
    fwrite(s,151);
    rightg = fread(s,1);
    fwrite(s,254);
    fwrite(s,152);
    leftru = fread(s,1);
    fwrite(s,254);
    fwrite(s,153);
    rightru = fread(s,1);
    if rightg>100 && rightg<200
        rgen = 1;
    else
        rgen = 0;
    end
    if leftg>100 && leftg<200
        lgen = 1;
    else
        lgen = 0;
    end
     if rightru>100 && rightru<200
         rru = 1;
     else
         rru = 0;
     end
     if leftru>100 && leftru<200
         lru = 1;
     else
         lru = 0;
     end
stateprev=state;
%Reads the controller BPCU.m
%(comment if distributed logic is used)
[c1,c2,c3,c4,c5,c6] = BPCU(rgen,lgen,rru,lru);
%If distributed control logic is used
%uncomment the following rows
%[c1,c2,c3] = BPCUAC(rgen,lgen);
%[c4,c5,c6] = BPCUDC(rru,lru);
if c1 ~= c1a || c2 ~= c2a || c3 ~= c3a ||...
   c4 ~= c4a || c5 ~= c5a || c6 ~= c6a
    t=1;
    statecount=statecount+1;
else
    t=0;
end
if stateprev ~=state
    disp('----------------------------------------')
    if rgen==0
        disp('rgen unhealthy')
    end
    if lgen==0
        disp('lgen unhealthy')
    end
     if lru==0
         disp('lru unhealthy')
     end
     if rru==0
         disp('rru unhealthy')
     end
     if rru==1 && lru==1 && rgen==1 && lgen==1
         disp('system runs normal')
     end
    disp('current state')
    disp(state)
    %If distributed control logic is used
    %uncomment the following row
    %disp(stateDC)
end
fwrite(s,254);
fwrite(s,26);
fread(s,1);
if c1 ~= c1a
    if c1==1
        fwrite(s,254)
        fwrite(s,15)
        fread(s,1);
    else
        fwrite(s,254)
        fwrite(s,7)
        fread(s,1);
    end
    fwrite(s,254)
    fwrite(s,23)
    c1a = fread(s,1);
end
if c2 ~= c2a
    if c2==1
        fwrite(s,254)
        fwrite(s,8)
        fread(s,1);
    else
        fwrite(s,254)
        fwrite(s,0)
        fread(s,1);
    end
    fwrite(s,254)
    fwrite(s,16)
    c2a = fread(s,1);
end
if c3 ~= c3a
    if c3==1
        fwrite(s,254)
        fwrite(s,11)
        fread(s,1);
    else
        fwrite(s,254)
        fwrite(s,3)
        fread(s,1);
    end
    fwrite(s,254)
    fwrite(s,19)
    c3a = fread(s,1);
end
 if c4 ~= c4a
     if c4==1
         fwrite(s,254)
         fwrite(s,14)
         fread(s,1);
     else
         fwrite(s,254)
         fwrite(s,6)
         fread(s,1);
     end
     fwrite(s,254)
     fwrite(s,22)
     c4a = fread(s,1);
```

```
 end
 if c5 ~= c5a
     if c5==1
         fwrite(s,254)
         fwrite(s,13)
         fread(s,1);
     else
         fwrite(s,254)
         fwrite(s,5)
         fread(s,1);
     end
     fwrite(s,254)
     fwrite(s,21)
     c5a = fread(s,1);
 end
 if c6 ~= c6a
     if c6==1
         fwrite(s,254)
         fwrite(s,12)
         fread(s,1);
     else
         fwrite(s,254)
         fwrite(s,4)
         fread(s,1);
     end
     fwrite(s,254)
     fwrite(s,20)
     c6a = fread(s,1);
 end
fwrite(s,254);
fwrite(s,37);
fread(s,1);
if t==1
    statetime(statecount)=toc;
else
    time(count)=toc;
    count=count+1;
end
```

```
end
disp('--------------------------------------')
disp('end')
%Close all relays
fwrite(s,254)
fwrite(s,25)
fread(s,1);
fwrite(s,254)
fwrite(s,0)
fread(s,1);
fwrite(s,254)
fwrite(s,1)
fread(s,1);
fwrite(s,254)
fwrite(s,2)
fread(s,1);
fwrite(s,254)
fwrite(s,3)
fread(s,1);
fwrite(s,254)
fwrite(s,4)
fread(s,1);
fwrite(s,254)
fwrite(s,5)
fread(s,1);
fwrite(s,254)
fwrite(s,6)
fread(s,1);
fwrite(s,254)
fwrite(s,7)
fread(s,1);
%Sound to know when the simulation stops
t1=1/10000:1/10000:2;
fadt1=(sin(2*pi*369.99*t1));
sound(fadt1,10000)
fclose(s)
delete(s)
clear s
```

B Component List

The main components used in the hardware testbed are listed in Table 3. The fuses, which are used by the testbed are of type 1 A 3AG fast acting fuses.

Quantity	Product
1	USB Relay Board 16-Channel 5 Amp DPDT 8 A/D Inputs
2	12 V Batteries
2	Power Inverter 12 VDC to 120 VAC
2	Transformers 120 VAC to 24 VAC
4	In Line Fuse Holder for 3AG Type
2	Rectifier Units 24 VAC to 0-5 VDC
2	2.4 V LED Diodes
2	24 VAC Lamp
2	Zener Diodes
2	DPDT Mechanical Switches
2	AA Batteries
1	AA Battery Holder

Table 3: List of the components used in the hardware testbed.